1995-03

Implementation of the SNR high-speed transport protocol (the transmitter part)

Mezhoud, Farah.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF THE SNR HIGH-SPEED TRANSPORT PROTOCOL (THE TRANSMITTER PART)

by

Farah Mezhoud

March 1995

Thesis Advisor: Gilbert M. Lundy

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Master's Thesis

4. TITLE AND SUBTITLE
Implementation of the SNR High-Speed Transport Protocol (the Transmitter Part)

5. FUNDING NUMBERS

6. AUTHOR(S)
Mezhoud, Farah

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to implement a transport protocol invented especially for high speed networking using the current workstations, so that the high throughput promised by the protocol will be achieved. The approach taken was to implement the SNR protocol, a transport protocol for high speed networking, named after its inventors, and composed of eight different machines (four transmitter and four receiver), using three Unix workstations connected with FDDI, allowing a throughput up to 100 Mbps.

This thesis is the implementation of the transmitter part of the protocol; the receiver part is done in parallel in a separate thesis. The four transmitter machines are implemented as four different Unix processes working in parallel and communicate through shared memory which provides the fastest means of exchanging information between processes. The protocol is implemented on top of the Internet Protocol layer using the "raw socket" as interface to access the IP facilities. The C programming language was used for the software implementation in order to access efficiently to the Unix system calls and thus reduce the overhead of the operating system.

This thesis shows that these new protocols can be successfully implemented using the current workstations and we expect that in a multiprocessor environment, where each machine is dedicated to a different processor, we will have even better performance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
SNR, protocol, implementation, high-speed, shared memory, raw socket	78
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Approved for public release; distribution is unlimited

IMPLEMENTATION OF THE SNR HIGH-SPEED TRANSPORT PROTOCOL (THE TRANSMITTER PART)

by

Farah Mezhoud
Major, Tunisian Army
Electronic Engineer, ESEAT France, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1995

Author: 

Farah Mezhoud

Approved By: 

Gilbert M. Lundy, Thesis Advisor

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is how to implement a transport protocol invented especially for high speed networking using the current workstations, so that the high throughput promised by the protocol will be achieved.

The approach taken was to implement the SNR protocol, a transport protocol for high speed networking, named after its inventors, and composed of eight different machines (four transmitter and four receiver), using three Unix workstations connected with FDDI, allowing a throughput up to 100 Mbps.

This thesis is the implementation of the transmitter part of the protocol; the receiver part is done in parallel in a separate thesis. The four transmitter machines are implemented as four different Unix processes working in parallel and communicate through shared memory which provides the fastest means of exchanging information between processes. The protocol is implemented on top of the Internet Protocol (IP) layer using the "raw socket" as interface to access the IP facilities.

The C programming language was used for the software implementation in order to access efficiently to the Unix system calls and thus reduce the overhead of the operating system.

This thesis shows that these new protocols can be successfully implemented using the current workstations and we expect that in a multiprocessor environment, where each machine is dedicated to a different processor, we will have even better performance.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Transmitting data at higher speeds has been the focus of many researchers for several years. As fiber optics is becoming more and more the media of choice to interconnect different types of computers, allowing the transmission of gigabits of information per second, network researchers and designers place more strain on the performance processing to match the data rates of fiber optic networks.

Fiber optics offers higher data transmission rates and lower error rates than copper wire, which justifies the need of change in design philosophy. New high speed protocols emphasize streamlining the normal data transmission processing for maximum throughput [Ref.19] and thus, increasing channel utilization in the presence of high speed, long latency networks.

The International Standards Organization (ISO) proposed a model for networks which has evolved into the ISO Open Systems Interconnection (OSI) [Ref. 20]. This model provides a definition of what each of seven network layers should be able to do. The relationship between these OSI seven layers is shown in Figure 1.

The transport protocol is considered by many network people as the keystone of the whole concept of a computer communications architecture [Ref. 18]. It provides the basic end-to-end service of transferring data between hosts. Any process or application can be programmed to access directly the transport services without going through session and presentation layers. Consequently, transport layer can be a source of processing overhead and may be responsible for the low throughput of the whole system since it has several critical functions related to data transfer [Ref. 3]. This includes detecting and correcting errors in received packets, usually through retransmission, providing a flow control mechanism for the system and delivering packets in order to the higher layers.

The performance of existing transport protocols is limited not only by the processor speed and memory access times but also by the processing overhead in the operating

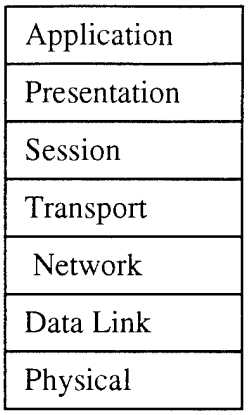


Figure 1: OSI Protocol Stack

system. The author in [Ref.18] suggested two approaches to overcome the difficulties encountered in transport protocols that led to low performance in presence of high speed networks.

The first approach is to adapt the conventional transport protocol to the new internetworking technology by improving its implementation. Research that has been done in this area show that the ability to provide high bandwidth to high speed network users depends heavily on protocol implementation [Ref.19]. The idea here is to take advantage of the long experience gained with the existing protocols and thus avoid certain implementation problems believed to be the cause of their slowness. Some of these problems are the transfer of data from the user, timer management, buffer management, connection state management, inter-process communication and scheduling.

The second approach is to invent new protocols devoted to be used in presence of high speed network.These new protocols, called lightweight transport protocols, are designed to offer higher throughput. Example of these protocols are VMTP (Vertasile Message Transaction Protocol), NETBLT (Network Bulk Transfer), XTP (Express Transfer Protocol) and SNR, named after its inventors (Sabnani, Netravali and Roome) from AT&T bell laboratories.

In order to provide the high throughput, VMTP uses selective retransmission for error recovery, groups packets into blocks and transmits large groups of packets in a burst as fast as the network allows, rather than using rate control. This method is found to be more efficient in processing cost [Ref. 3]. NETBLT also groups packets into blocks, uses a rate control scheme (packet per second) based on the network congestion and selective retransmission is used for error recovery. XTP on the other hand is designed to be implemented in hardware and it combines the transport layer with the network layer. Flow control is achieved through the use of parameters which provide visibility of the receiver's buffer to the transmitter. It also uses rate control and the selective repeat method for error recovery.

B. OBJECTIVES

In the Naval Postgraduate School, more and more attention has been given to these new protocols. In particular, Lundy and Tipici conducted research in 1993 on the SNR protocol [Ref. 3]. They used a System of Communicating Machines (SCM) to analyze the protocol and verify that it is free from logical errors. In their conclusion they posed the question whether this protocol efficient enough to provide the high throughput which is expected from the lightweight transport protocols.

This thesis is a continuing work on this transport protocol and an attempt to answer this question by implementing the transmitter part on the current workstations and performing realistic performance tests. The implementation of the SNR receiver is being conducted in parallel in a separate thesis by W. J. Wan [Ref. 22]. Our major goal in this work is to optimize the implementation in order to reduce the processing overhead and thus achieve the high throughput, promised by this protocol. A third research is conducted in parallel by R. Grier [Ref. 23] to test and verify our SNR implementation.

C. ORGANIZATION OF THE THESIS

In this implementation we follow very closely the specification introduced in [Ref. 3]. In order to make this thesis independently readable, the state diagram and the Predicate

Action Table of each of the SNR machines are reproduced in this thesis. Some corrections and implementation details are added to the protocol and will be pointed out as necessary.

The thesis is organized into four chapters. Chapter II discusses the problems with existing protocols and introduces the specification of the SNR protocol (transmitter and receiver), including the state diagram and Predicate Action Table of each machine. Chapter III discusses the approach taken to implement the transmitter part, the algorithm of each transmitter machine is given. Finally we conclude in Chapter IV, along with some recommendations and future work that can be done as continuation on the same subject.

II. SNR PROTOCOL

A. INTRODUCTION

1. Problems with Existing Protocols

In this section we will discuss briefly some of the problems that are considered to be the most important cause of the under utilization of the full potential offered by the fiber optic technology and therefore must be avoided when designing lightweight protocols.

Existing protocols were designed when the processing speed of the CPU's was higher than the bit rates provided by the media and for that reason, one of the important consideration of their design was not to saturate the transmission media with high data rates. Today, with the introduction of fiber optic technology, the problem is reversed and the bottleneck has moved to the communications processing part of the system.

Protocols must be interfaced with the host operating system, a required support to move data from one host to another (interrupts, I/O devices, buffers...). Some suggestions are made [Ref. 21] to get the best support from the operating system:

- Parallel processing of independent functions of the protocol,
- Avoiding the movement of data in memory, since this is the most costly operation in packet processing,
- Making minimal use of operating system timer package.

The overhead of managing timers can sometimes be a burden.Timers are used to recover from channel losses. In a positive acknowledgment scheme, there must be a timer associated with every packet. A timer must be set, monitored, cleared and reset every time a packet is sent or received.

Another problem associated with timers is the values that this timers must be set to, which they depend on the round trip delay (RTD). Often the RTD can not be predicted with accuracy since it varies with the condition of the networks.

When slow underlying conventional networks were used, the major issue was not to overflow channels and for that reason, variable size packets, that are just large enough to fulfill the need, were used, increasing the processing time at the receiver due to decoding operations. This was not a problem since the bit rates in the network was slow anyway. But with high speed networks, trying to minimize the packet size is just a wasted effort and for this reason standard packet format are used.

The Go-Back-N method for error recovery is another defect of current protocols. Whenever the receiver detects a loss of a packet, it sends a negative acknowledgment message (NAK) to the transmitter, requiring retransmission of all the packets after the last correctly received one. If the data rates are high or the retransmission channel is long, this method may require many good packets to be retransmitted, causing significant loss of throughput.

A conservative flow control scheme which uses the sliding window technique may limit the throughput of the protocol in long-delay situations [Ref. 18]. This is because the sliding window does not decuple acknowledgments from flow control. In the transmitter side, the window size is increased only by acknowledgments from the receiver. On the other hand, if the first packet of a given window gets lost, the receiver can not acknowledge any of the rest of the packets and has to wait for the lost packet to be recovered. At the same time, we can have a situation where the transmitter also has to wait for its window to be increased.Therefore, both transmitter and receiver are waiting for the retransmitter timer to expire. A better solution is to separate this two functions.

2. Introduction to the SNR Protocol

The SNR protocol was first introduced in [Ref. 12], then in [Ref. 2] a formal specification was given using the System of Communicating Machines (SCM) model, and finally in [Ref. 3] a further study was given on the protocol to refine and improve the SCM specification by applying the associated system analysis. More details about the protocol can be found in [Ref. 3] from which graphs and tables are reproduced in this chapter.

The SNR protocol requires a full duplex link between two host computers connected across a high speed network as shown in Figure 2.

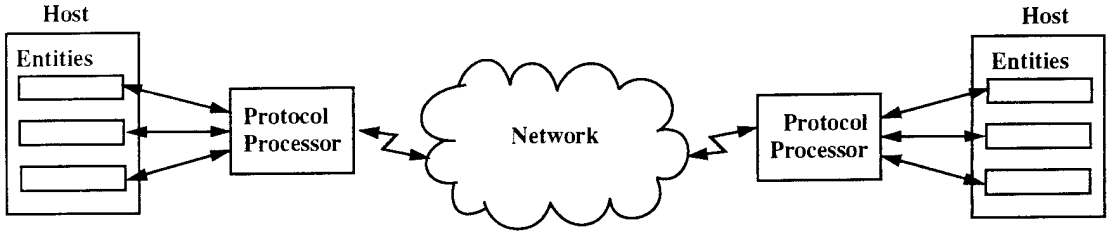


Figure 2: Network,Hosts,Entities and Protocol Processors

Each host system in the network consists of eight finite state machines (FSM), four for the transmitter and four for the receiver. Each machine in the protocol has a specific function in coordination with the other machines

B. DESIGN AND ORGANIZATION

To overcome the difficulties of the current transport protocols, discussed earlier in this chapter, some design principles are followed when designing the SNR protocol in order to simplify the protocol and reduce the processing overhead of the operating system.

Unlike most current protocols where status messages are exchanged only when certain events occur, in the SNR protocol full state information is exchanged between transmitter and receiver on a regular basis and therefore, the generation of acknowledgments is totally decoupled from the data traffic. Although this technique increases the number of packets transmitted, which is negligible compared to the very high bandwidth of the fiber and the speed up it provides in processing, it has the advantage of simplifying the protocol and allowing parallel protocol processing which leads to higher performance.

In order to avoid wasting the network resources caused by the retransmission of thousands of good packets when using the Go-Back-N error recovery method, the SNR

protocol uses selective repeat method of retransmission, and the concept of blocking. Blocking reduces the overhead of maintaining large tables and complex procedures that are required for the selective repeat method. A group of packets (typically 8) is called a block. Packets within the block are transmitted and treated separately by the network. Upon successful reception of all of the packets in a block, the receiver acknowledges the block, rather than the individual packets. If a packet in a block gets lost, then the whole block of packets is retransmitted. Despite the retransmission of some good packets in the block, this method is used in order to simplify the protocol and thus gain in processing speed knowing that fiber-optic media supporting the protocol has very low error rates.

A very important idea used in designing the SNR protocol that contributes significantly to the improvement of the performance, is the concurrent execution of several independent functions of the protocol. The four machines on each side (transmitter and receiver) execute in parallel with small interaction between them.

Flexibility is another key of the SNR protocol and it is illustrated by specifying three modes of operation:

Mode 0 is the simplest mode and has no error control or flow control. It is suited for virtual circuit networks and for the cases where quick interaction between the communicating entities is desired and short packets are used.

Mode 1 has no error control but provides flow control. It is suitable for real time applications where error control is not needed and packet sizes are small.

Mode 2 has both error control and flow control. It is useful for large file transfers in all types of network services.

The way flow control is done in SNR protocol is as follows:

The receiver writes the available buffer space it has in units of blocks into the *buffer-available* field of the receiver control packets, and the transmitter maintains the variables L, the maximum window size in units of blocks which will be chosen slightly larger than (RTD * maximum bandwidth)/number of bits in a block, and NOU, the number of outstanding blocks which have been transmitted but not yet acknowledged. Every time

the transmitter completes the transmission of a block of packets, it increments NOU, and every time it receives a receiver control packet it decrements NOU by the number of acknowledged blocks. It starts the transmission of another new block only if NOU is less than L and *buffer-available* is greater than NOU.

C. PROTOCOL SPECIFICATION

The general organization of the eight machines, including global variables through which machines communicate, is shown in Figure 3. Details of implementing the transmitter part will be left to the next chapter.

1. Transmitter Processes

a. Process T1

Transmission of new data packets and retransmission of unacknowledged packets is done by machine T1. The state diagram and the Predicate Action Table of T1 are shown in Figure 4.

Machine T1 starts executing its functions when the global variable T_active is set to TRUE by machine T2 upon successful connection establishment. In mode 0, T1 transmits data as long as T_active remains TRUE and there is data in the buffer OUTBUF. Every time the transmission of a block is completed, it increments the variable UWt (transmitter upper window).

In mode 1 and in order to provide the flow control, a new block is transmitted only if the receiver has space in its buffer and the maximum window size has not been exceeded. The block is transmitted if NOU < L and buffer_available - NOU > 0. NOU is incremented every time a complete block is transmitted. It is decremented by machine T2 by the number of acknowledged blocks read from the receiver state information packet.

In mode 2 if the "count" field of any block in the LUP table becomes 0, then T1 stops transmitting new packets and retransmits all packets in the expired block, count is

reinitialized to 0 when the whole block is retransmitted. The variables UWt and NOU are incremented and a new entry in LUP table is made every time the transmission of a new block is completed.

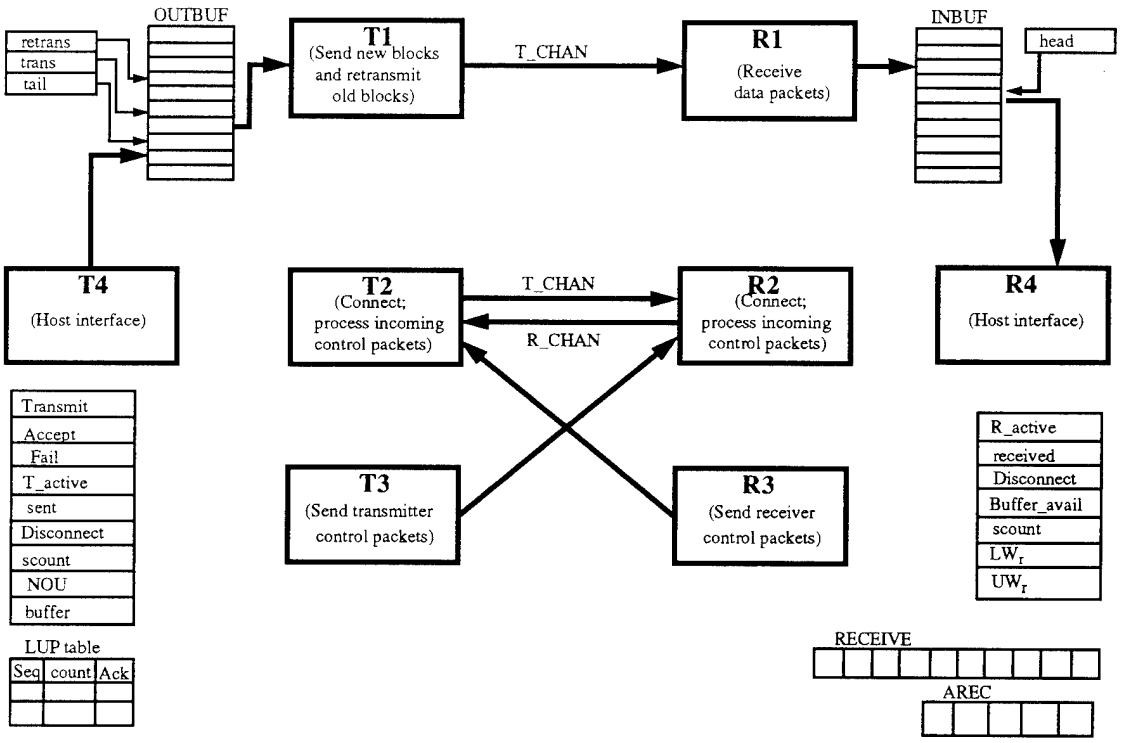


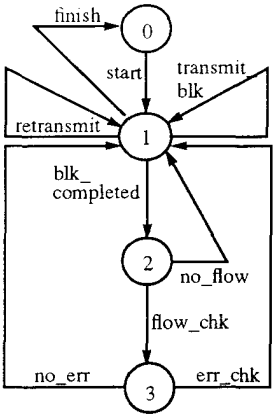
Figure 3: Machine Organization Including the Shared Variables

b. Process T2

Process T2 has to establish the connection, receive and process the receiver control packets and finally terminate the connection. State diagram and Predicate Action Table are shown in Figure 5.

Process T2 begins when the global Transmit is set to TRUE by T4. It then sends a "conn_req" packet with negotiated parameters and receives the answer "conn_ack"

packet with the same parameters set by the receiver, and if they agree on these parameters, it sends a "con_conf" packet to the receiver to confirm the connection and sets the global variable T_active to TRUE, allowing the other transmitter machines to begin executing in



Transition	Predicate	Action
start	T_active=T	null
finish	T_active=F	null

retransmit	T_active=T ^
	mode= 2 ^ **Expired**(LUP) /=0
		Packet.seq:=(**Expired**(LUP)-1)*block_size+
		retrans_count;
		Packet.data:=OUTBUF(Packet.seq mod
		OUTBUF'length);
		Enqueue(Packet,T_CHAN);
		sent:= T;
		inc (retrans_count);
		if retrans_count > block_size then
		retrans_count:=1;
		LUP((**Expired**(LUP)-1) mod L+1).count:=
		initial value;
		end if;

transmit_
blk
	T_active=T ^
	not (**Empty**(OUTBUF)) ^
	trans_count <= blk_size ^
	(mode=0 v ((NOU < L ^
	buffer - NOU>0)^
	(mode=1 v **Expired**(LUP)=0)))
		retrans_count := 1;
		Packet.seq:=UW_t * block_size +
		trans_count;
		Dequeue(Packet.data,OUTBUF);
		Enqueue(Packet,T_CHAN);
		sent:=T;
		inc (trans_count);

blk_ completed	trans_count > blk_size	trans_count := 1; inc (UW_t);
no_flow	mode = 0	null
flow_chk	mode=1 v mode = 2	inc (NOU);
no_err	mode = 1	null
err_chk	mode = 2	**Insert** (UW_t, LUP);

Figure 4: T1 State Diagram and Predicate-Action Table

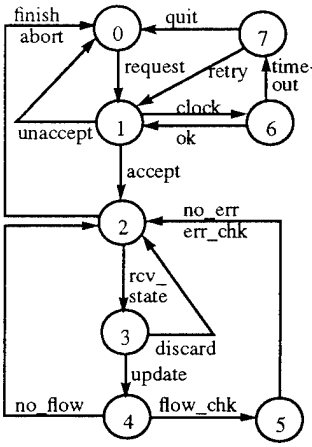
parallel. The connection is terminated by sending a "disc" packet when the global variable Transmit is set to FALSE by T4. During data transfer, T2 receives the receiver control packets and depending on the mode of operation, the following actions are taken:

- Scount is reset to 0 (more explanation of the use of this variable in T3)
- NOU is incremented by the number of acknowledged blocks (mode 1 and mode 2)

• LWr is copied into LWt, LOB bit field is copied into HOLD, and buffer-available field of the receiver control packet is copied into buffer-available (mode 1 and 2)

• Acknowledged packets are removed from OUTBUF and from LUP table

• count field in LUP table of the unacknowledged blocks are decremented by the number of the received k.

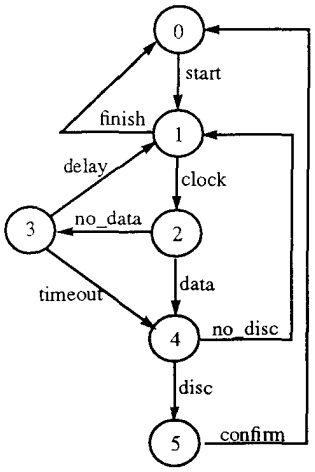


Transition	Predicate	Action
request	Transmit=T ^ Accept=T ^ Fail=F	**Enqueue** (*Conn_Req*, T_CHAN);
accept	R_CHAN(front) = *Conn_Ack* ^ **Acceptable** (R_CHAN(front))	T_active := T; **Enqueue** (*Conn_Conf*, T_CHAN); **Dequeue** (R_CHAN);
unaccept	R_CHAN(front) = *Conn_Ack* ^ not (**Acceptable** (R_CHAN(front)))	Accept:=F; **Dequeue** (R_CHAN);
clock	**Empty** (R_CHAN) ^ *clock_tick*	inc (delay);
ok	delay < reset	null
timeout	delay = reset	inc (attempts); delay:=0;
retry	attempts < max_attempts	**Enqueue** (*Conn_Req*, T_CHAN)
quit	attempts = max_attempts	Fail := T
finish	Transmit = F ^ **Empty** (OUTBUF) ^ Disconnect = F ^ ((mode = 2 ^ **Empty** (LUP)) v mode = 1 v mode = 0)	T_active:=F; **Enqueue** (*Disc*, T_CHAN);
abort	Disconnect = T	T_active:=F; Transmit:=F;
rcv_state	not (**Empty** (R_CHAN)) ^ Disconnect=F	null
discard	R_CHAN(front).seq <= high v R_CHAN(front)=*Conn_Ack*	**Dequeue**(R_CHAN);
update	R_CHAN(front).seq > high	scount:=0; high:=R_CHAN(front).seq;
no_flow	mode = 0	**Dequeue**(R_CHAN);
flow_chk	mode = 1 v mode = 2	**Balance**(R_CHAN(front).LOB,HOLD, R_CHAN(front).LW_r, LW_t, NOU); HOLD:= R_CHAN(front).LOB; LW_t := R_CHAN(front).LW_r; buffer := R_CHAN(front).buffer_avail. **Update_outbuf** (OUTBUF, LW_t);
no_err	mode = 1	**Dequeue** (R_CHAN);
err_chk	mode = 2	**Update_LUP** (LUP, HOLD, LW_t, R_CHAN(front).k); **Dequeue** (R_CHAN);

Figure 5: T2 State Diagram and Predicate-Action Table

c. Process T3

T3 starts when T_active is set to TRUE by machine T2 and executes its function each time a clock-tick occurs. Details of implementing the clock-tick are left to the next chapter. State diagram and Predicate Action Table of T3 are shown in Figure 6.



Transition	Predicate	Action
start	T_active=T	null
clock	*clock_tick* ^ T_active = T	inc (scount)
no_data	sent=F	inc (count)
delay	count < k ^ scount < Lim	null
timeout	count = k v scount >= Lim	**Enqueue** (*T_state*,T_CHAN); k:= min (2*k, klim)
data	sent = T	**Enqueue** (*T_state*, T_CHAN); k:= 1
no_disc	scount < Lim	sent:= F; count:= 0
disc	scount = Lim	Disconnect:= T
confirm	T_active = F	null
finish	T_active = F	null

Figure 6: T3 State Diagram and Predicate-Action Table

The main function of process T3 is to transmit periodically the transmitter control packets. In addition, and if no receiver control packets are received for a predetermined amount of time, it initializes an abnormal connection termination. For this purpose,it increments Scount every time it transmits a control packet, and T2 sets this variable to 0 every time it receives a receiver control packet. If Scount ever reaches a predetermined value Lim, then T3 sets the variable Disconnect to TRUE which causes T2 to send a "disc" packet and sets T_active and transmit to FALSE.

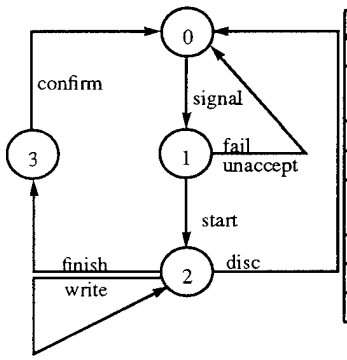
T3 transmits control packets depending on the global variable Sent, which is set to TRUE by process T1 after every data packet transmission.

If Sent is TRUE, a control packet is sent and the variable Sent is reset to FALSE. If Sent is FALSE, then k is recalculated using the formula *k=MIN (2*k, k*Lim)* and control packet transmission is delayed for *k*Tin* seconds, where Tin is the period of the

clock-tick mechanism. If meanwhile Sent becomes TRUE, then T3 stops waiting and transmits the packet and resets k to 1.

d. Process T4

T4 is the first process invoked and is the interface between SNR protocol and the upper layer (the host). When it is invoked, it sets Transmit to TRUE, causing process T2 to initiate the connection. The result of this phase is known through the global variables T_active, Fail and Accept. If Fail is TRUE or Accept is FALSE,T4 sets Transmit to FALSE and the protocol is stopped, and the host is notified of the failure, otherwise the connection is established and data are transmitted between the two hosts. State diagram and Predicate Action Table of T4 are shown in Figure 7.



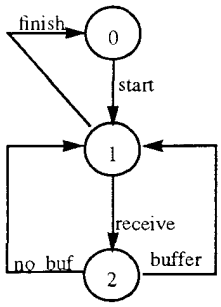
Transition	Predicate	Action
signal	*transmission signal from the host*	Transmit := T; Accept := T
fail	Fail = T	Transmit := F; *notify host of failure to connect;*
unaccept	Accept = F	*notify host of unacceptable connection*
start	T_active = T	null
write	not (**Full** (OUTBUF)) ^ not(eot) ^ T_active=T	**Enqueue** (*data stream from the host*, OUTBUF)
finish	eot ^ T_active = T	Transmit := F
confirm	T_active = F	*notify host of completion*
disc	T_active = F	*notify host of disconnect*

Figure 7: T4 State Diagram and Predicate-Action Table

2. Receiver Processes

a. Process R1

R1 removes the received data packets from T_CHAN and either inserts them into the buffer INBUF (mode 1 or 2) or passes them to the host directly (mode 0). It starts its function when R_active is set to TRUE by machine R2. State diagram and Predicate Action Table are shown in Figure 8.



Transition	Predicate	Action
start	R_active = T	null
finish	R_active = F ^ **Empty** (INBUF)	null
receive	T_CHAN (front) = *DATA*	null
no_buf	mode = 0	*Pass* T_CHAN(front) *to the host;* **Dequeue** (T_CHAN)

buffer	mode = 1 v mode = 2	**Order_insert**(T_CHAN(front), INBUF,
		RECEIVE, LW_r, duplicate);
		if not duplicate then
		received := T;
		Process_packet (T_CHAN(front).seq, RECEIVE,
		AREC,Buffer_avail,LW_r,UW_r,LOB);
		end if;
		Dequeue (T_CHAN);

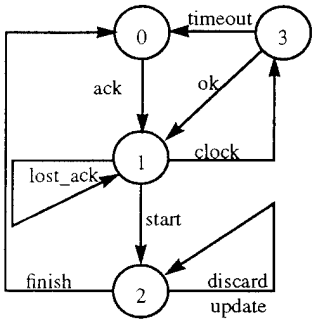
Figure 8: R1 State Diagram and Predicate-Action Table

b. Process R2

Process R2 establishes the connection with the transmitter and thereafter receives and processes the transmitter control packets. It starts its function upon reception of a Conn_req message. Process R2 is the receiver counterpart of process T2. State diagram and Predicate Action Table are shown in Figure 9.

c. Process R3

State diagram and Predicate Action Table of process R3 are shown in Figure 10. It has exactly the same structure and function as the process T3: it transmits the receiver control packets periodically to the transmitter through R_CHAN and initiates an abnormal connection if no transmitter control packets are received for a predetermined amount of time.



Transition	Predicate	Action
ack	T_CHAN (front) = *Conn_Req*	**Evaluate** (*Conn_req*); **Dequeue** (T_CHAN); **Enqueue** (*Conn_ack*, R_CHAN);
clock	*clock_tick* ^ **Empty** (T_CHAN)	inc (delay)
ok	delay < reset	**Enqueue** (*Conn_ack*, R_CHAN);
timeout	delay = reset	null

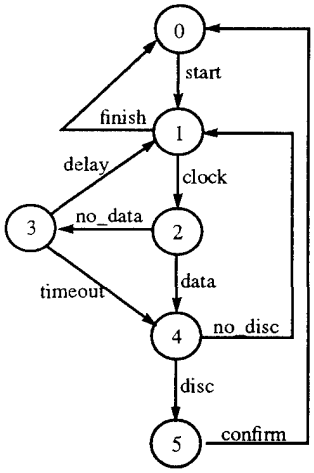
start	T_CHAN(front) = *Conn_conf* v
	T_CHAN(front) = *T_state* v
	T_CHAN(front) = *Data*
		R_active:= T;
		if T_CHAN(front) = *Conn_conf* then
		Dequeue(T_CHAN);
		end if;

finish	Disconnect = T v T_CHAN(front)=*Disc*	R_active := F;

update	T_CHAN(front) = *T_state* ^
	T_CHAN(front).seq > high
		scount := 0;
		high := T_CHAN(front).seq;
		Dequeue (T_CHAN);

discard	(T_CHAN(front) = *Conn_conf* v T_CHAN(front) = *Conn_req*) v (T_CHAN(front) = *T_state* ^ T_CHAN(front).seq <= high))	**Dequeue** (T_CHAN);
lost_ack	T_CHAN(front) = *Conn_req*	**Dequeue** T_CHAN); **Enqueue** *(Conn_ack*, R_CHAN)

Figure 9: R2 State Diagram and Predicate-Action Table



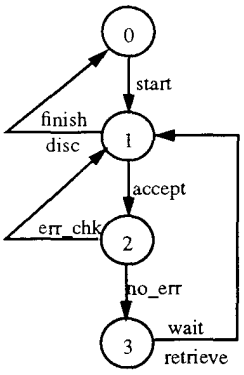
Transition	Predicate	Action
start	R_active=T	null
clock	clock_tick ^ R_active=T	inc(scount)
no_data	received=F	inc(count)
delay	count<k ^ scount<Lim	null
timeout	count=k v scount>=Lim	**enqueue**(R_state,R_CHAN); k:=min(2*k, klim)
data	received=T	**enqueue**(R_state, R_CHAN); k:=1
no_disc	scount<Lim	received:=F; count:=0
disc	scount=Lim	Disconnect:=T
confirm	R_active=F	null
finish	R_active=F	null

Figure 10: R3 State Diagram and Predicate-Action Table

d. Process R4

Process R4 is the interface to the receiving host. It passes the data in INBUF to the host and notifies it of any errors that occur. State diagram and Predicate Action Table of process R4 are shown in Figure 11.

More details of the receiver processes implementation can be found in [Ref. 22].



Transition	Predicate	Action
start	R_active = T	null
finish	R_active = F ^ **Empty** (INBUF) ^ Disconnect = F	null
disc	Disconnect = T	*notify host of disconnect*
accept	Disconnect = F ^ not (**Empty**(INBUF)) ^ mode=1 v mode=2 ^ *signal from host*	null
no_err	mode = 1	null
wait	**Wait** (INBUF, RECEIVE) = T	null
retrieve	**Wait** (INBUF, RECEIVE) = F	**Retrieve_mode1** (INBUF,RECEIVE, AREC,buffer_avail,LW_r,UW_r,LOB);
err_chk	mode = 2	**Retreive_mode2** (INBUF,RECEIVE, buffer_avail);

Figure 11: R4 State Diagram and Predicate-Action Table

III. IMPLEMENTATION OF THE TRANSMITTER

A. GENERAL APPROACH

1. Introduction

The software implementation of the protocol will be done on the Naval Postgraduate School FDDI (100 Mbps) network, which is composed of three workstations with two different versions of the Unix operating system, one IRIX and two SOLARIS system.

Consequently, the portability of the resulting product is an important issue that must be taken into account from the beginning. Protocols are implemented at the low level of the operating system (the kernel level) to take advantage of the facilities and functions available at that level, which are sometimes different from one version to another. Moreover, this is not considered in the specification of the protocol and left as abstraction.

As mentioned in early chapters, minimizing the overhead of the operating system in order to speed up the overall execution of the software is our main goal and for that reason we chose to use the C programming language to help achieving this goal by allowing more flexibility to access the operating system functions with efficiency. It is easier to execute system calls with minimum cost, i.e, minimum number of instructions. In addition it helps developing portable code.

Dealing efficiently with the inevitable overhead caused by the operating system related functions is the most important implementation issue in this thesis.

On the other hand, the basic idea of the design of the SNR protocol is to have four different machines (processes) on the transmitter side that work in parallel, and other four processes on the receiver side. Since the number of processors in the workstations used to implement this protocol is less than the number of processes executing in parallel, expensive operations, such as context switches to save the current status of the CPU during scheduling are needed. In fact, the scheduler on the Unix system belongs to the general class of operating system schedulers known as round robin with multilevel feedback,

meaning that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds its time quantum, and feeds it back into one of several priority queues [Ref. 9]. A process may need many iterations through the "feedback loop" before it finishes, therefore, a lot of overhead is involved, slowing down the execution of the program.

Context switches are also necessary to handle the clock tick event used by both machine T2 during the connection establishment and by machine T3 to execute its function, as will be explained later in this chapter.

The SNR protocol will be implemented on top of the Internet Protocol (IP), a connectionless protocol that has been developed for the network layer. Therefore an interface is needed to access the IP facilities. An SNR header is added to the packet before it is given down to the IP protocol, which has the responsibility for routing and relaying packets in an internet environment.

Each protocol has to have an identifier, so the IP layer knows what protocol it deals with. We chose to assign **191** as the SNR identifier.

2. The Raw Socket

The Unix operating system provides the socket interface to support communication protocols and creates an endpoint for communication. The types of socket currently defined are:

- stream socket: provides sequenced, reliable, two-way connection based byte streams.
- datagrams socket: supports datagrams (connectionless, unreliable messages of a fixed maximum length).
- sequenced packet socket: provides a sequenced, reliable, two-way connection based data transmission path for datagrams of fixed maximum length.
- RDM socket: not implemented yet.

• raw socket: available only to the super user and provides access to the internal network interface, generally used to develop new protocols.

The first two types are designed to support the Internet TCP and Internet UDP protocols respectively.

In the current implementation of the SNR protocol we chose to build the interface to the underlying protocol layer (the IP protocol) by using the raw socket, since it is designed to support developing new protocols. It will be used to implement the transmitter channel (T_CHAN) mentioned in the specification and left as abstraction. The system call "sendto" is used in this case. To implement the receiver channel (R_CHAN) a circular buffer is used, and packets received from the raw socket, using the system call "receivfrom", are enqueued in this buffer and later processed by T2 during the connection establishment and by T3 to receive the receiver state packets, as will be shortly explained.

3. Shared Memory

Since the implementation of the SNR transmitter, as well as the receiver, involves the interaction of four different processes, we have to look carefully at the different methods that are available for different processes to communicate with each other.

In traditional single process programming, different modules within the single process can communicate with each other using global variables, functions calls and the arguments and results passed back and forth between functions and their callers. In our case instead, we deal with separate processes, executing in parallel, each with its own address space and hence, there are more details to consider.

The unix system supports three types of Inter-Process Communication (IPC): messages, semaphores and shared memory.

Messages allow processes to send formatted data streams to an arbitrary process. Data is transmitted between processes in discrete portions called messages.

Shared memory allows processes to share parts of their virtual space. This is done by allowing processes to set up access to a common virtual memory address space, and the sharing occurs on a segment basis, which is memory management, hardware dependent.

Semaphores allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. Communication is made through the exchange of semaphore values. As a form of Inter-Process Communication, they are not used for exchanging large amounts of data.

In this current implementation of the SNR transmitter we will use the shared memory mechanisms along with semaphores to synchronize the access of different machines to some global variables that constitute critical sections, which we will detailed later in this chapter.

This sharing of memory provides the fastest means of exchanging information between processes since data is not moved at all. A segment is first created outside the address space of any process, and then each process that wants to access it, executes a system call to map it into its own address space. Several hardware segmentation registers are used to address data segments, and the system keeps one or more of these registers free to make this mapping very fast. Since the shared segment is within the process's address space, access to it is just as fast as access to local variables.

4. Timing Mechanism

As mentioned in the protocol specification [Ref. 3], the timing mechanism is based on an event, called "clock-tick" and occurs periodically at T_{in} second intervals. To implement this mechanism we use the interval timer facility provided by unix system. This facility is a powerful tool that provides the microsecond resolution[1] and allow the user to specify both an offset from the current time (the delay), and the recurrence time (the interval). The timer will not fire until the delay has passed, and then will continue to fire at the end of each interval.

1. in fact,resolution is machine dependent and limited by the interval clock of the machine.

This timer is used by both machine T3, to send transmitter control packet every T_{in} second and by machine T2 while establishing the connection. T_{in} is given by the formula: $T_{in} = max(RTD/kou,IPT)$, where RTD is the estimated round trip delay for the logical connection, the constant *kou* is typically a power of two, such as 16 or 32, and IPT is the average time between two data packet transmission. RTD is computed during the connection establishment using the "gettimeofday" system call just before sending a request to get the old date and just after receiving the acknowledge of the same request to have the new date. The difference between old and new date is equal to RTD.

5. Using Signals

The way the timing mechanism, explained in the previous paragraph, works, is by sending a signal called "SIGALRM" each time the delay, specified in the system call has passed. The process using the timer, needs to arrange and catch this signal.

Beside SIGALRM, two other signals are used in the current implementation of the SNR transmitter as follows:

- SIGUSR1: used by machine T2 to terminate T2_slave process that is responsible for receiving packets from the raw socket.

- SIGUSR2: sent from T2 to process T4 to inform the latter about the end of the connection establishment phase. By this way we avoid the busy waiting of T4.

B. TRANSMITTER MACHINES

1. Time Dependency

Machine T4 is invoked the first by signal from the upper layer (when the user needs a service). It then gets the shared memory identifier and attaches it to its address, initializes the global variables, including the negotiated parameters, forks[2] T2 and then waits the

2. fork-exec is the way processes are created in UNIX system.First a process forks,meaning that it makes a copy of itself,and then by exec system call,this copy will be replaced by the process we want to execute.

arriving of the signal SIGUSR2, sent by T2 at the end of the connection phase. T2 begins by forking T2_slave, the fifth process added to the protocol to receive packets from the raw socket, enables the timer to fire every T_{in} microsecond (T_{in} is first chosen to be 50 milliseconds) and then executes the standard three way hand-shake connection establishment with the receiver, which we will detail later. At the end of this procedure, it sends the signal SIGUSR2 to T4, which depending on the result of the connection phase, terminates the session by notifying the host or, forks the remaining machines, T1 and T3, and from that point in time, all machines begin executing in parallel following their specification. Figure 12 shows the time dependency between machines invoked in the transmitter part of the SNR protocol.

2. Critical Sections and Synchronization

By using the shared memory for communication between processes, the speed to access data is guaranteed to be very fast, as explained earlier in this chapter. However, at the same time we introduce a new problem of critical section and data consistency, and therefore, a way to synchronize the access to global variables is needed. However, this problem arises only when more than one process wants to change the value of the same variable.These common sources will be controlled by semaphores and before a process can obtain that resource and change its value, it needs to test the current value of the semaphore, which is stored in the kernel, and depending on that value, it can access that variable with the guarantee that no other process can interfere with it until it will be released, or it waits until it will be freed.

In Table 1, global variables are shown along with processes that access to them. Column "set by " is used to determine the resource, accessed by more than one process, and thus needs to be controlled by semaphore.

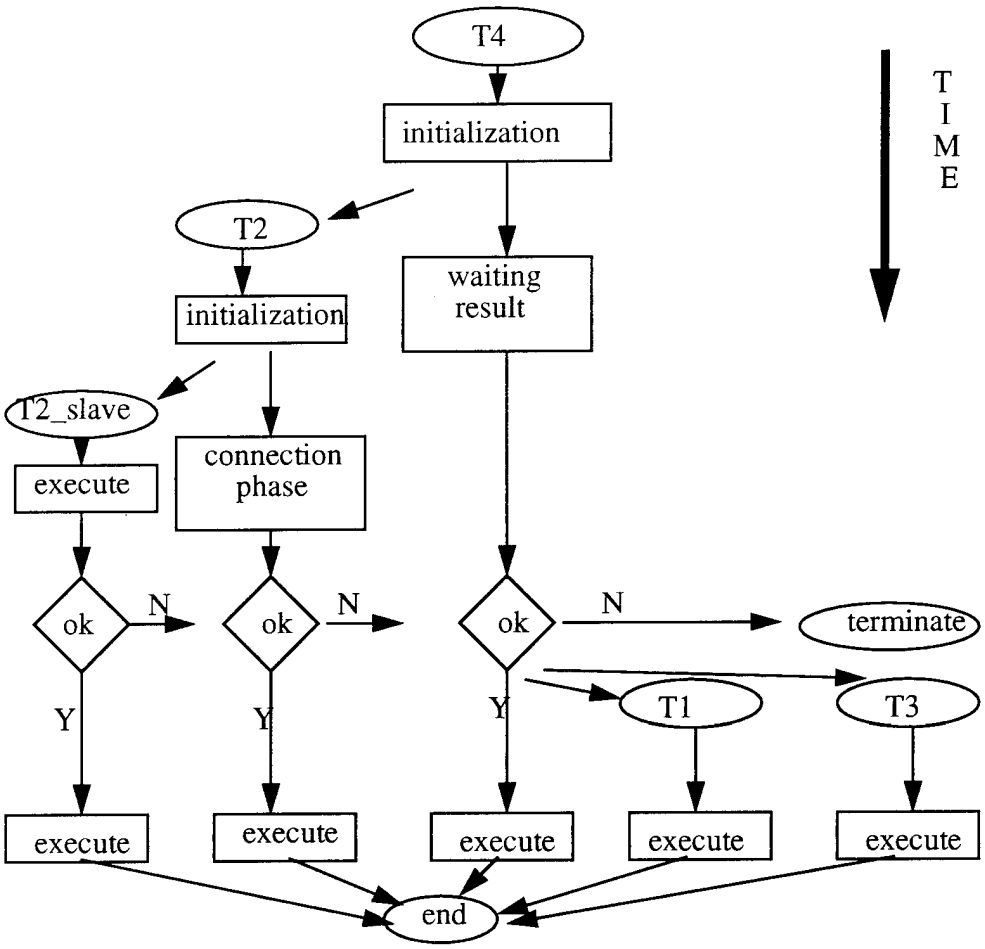


Figure 12: Timing Dependency

As can be seen in Table 1, we need four semaphores for the variables: Sent, Scount, NOU and LUP. Although Transmit is accessed by both T2 and T4, we do not need a semaphore because it is set to TRUE only once at the beginning by T4, and later if it is necessary, it will be set only to FALSE, either by T2 or T4.

3. SNR Packets Format

Transmitter and receiver control packet format and data packet format are detailed in [Ref. 3] and in this thesis we use these same three formats and add another four types of packets needed to implement the protocol: Connection request packet, connection acknowledge packet, connection confirmation packet and disconnection type packet. In addition we define the SNR header to be as follows: LCI, type, block-no, packet-no within the block and data_length. Table 2 shows the SNR header format. The header will be the same for all seven types of packets. The sequence number defined in [Ref. 3] is divided into block-no and packet-no. Data_length (16 bits) was not defined in the original specification and added in this thesis to allow the transmission of different types of files including binary files. When dealing with binary files it is difficult to define a special character indicating the end of the file, since in this protocol the data packet length is fixed and consequently sometimes padded characters are added to the last data packet. This new field allows the receiver to separate padded characters from the original data. Also in this implementation, we always send an exact number of blocks, and consequently, we sometimes add empty packets to complete a block. The data_length for these empty packets will be zero (0).

SNR_HDR: common to all types of packet, Table 2 shows the format of the SNR header and in Figure 13, its C definition is shown.

LCI is the logical connection identifier, which is a unique sequence number across all the logical links which the host computer is engaged in. The type field identifies the type of the packet and ranges from 0 through 6. Block number used only for type 2 (data packet) and contains the number of the block being transmitted (received). Packet number is used for all seven types and contains the packet number (for data type packet, contains the number of the packet within the block).

	set by	read by
Transmit	T2,T4	T2
Accept	T2	T2,T4
Fail	T2	T2,T4
T_active	T2	T1,T3,T4
Sent	T1,T3	T3
Disconnect	T3	T2
Scount	T2,T3	T3
NOU	T1,T2	T1
Buffer	T2	T1
TAIL	T4	T1,T2,T4
TRANS	T1	T1,T2,T4
RETRANS	T2	T4
LUP	T1,T2	T1

Table 1: Accessing Global Variables

LCI	type	block-no	packet-no	data_length

Table 2: SNR Header Format

```
typedef struct {
        u_char    block-no;      // 8 bits
        u_char    packet-no;     // 8 bits
        } SEQ;
struct SNR_HDR {
        u_char    lci;           // 8 bits
        u_char    type;          // 8 bits
        SEQ       seq;
        u_short   data_length;   // 16 bits
        };
```

Figure 13: SNR Header Definition

Receiver control packet (R_state): contains the state information of the receiver. Table 3 shows the format of the receiver control packet.

SNR_HDR	k	LWr	Buffer-avail	LOB	Error-check

Table 3: Receiver Control Packet (type 0)

```
struct rstate {
        int       k;
        int       LWr;
        int       Buffer-available;
        u_char    LOB[size_LOB];
        int       Error-check;
        };
```

Figure 14: C Definition of the Receiver Control Packet (type 0)

The sequence number of the packet is given by packet-no field of the SNR header and block-no is not used in this case; k, LWr, Buffer-avail and LOB contain values of receiver variables just prior to the transmission of the packet. The variable k is the interval between two control packet transmissions of the receiver in unit of T_{in}, LWr is a block sequence number such that all the blocks with sequence numbers less than this have been correctly received and acknowledged, Buffer-avail is buffer space available at the receiver in units of blocks, and LOB is a bit map representing the outstanding blocks between LWr and (LWr + L - 1) where L is the maximum window size. The first bit of LOB corresponds to LWr and is always set to 0. The other bits are set to 1 if the corresponding blocks have been received correctly, otherwise they are set to 0. LWr and LOB fields are used together to acknowledge the blocks received correctly. The last field contains an error detection code.

Transmitter control packet (T_state): contains the state information of the transmitter. The packet format is shown in Table 4 and in Figure 15 a C definition of the same packet is given.

SNR_HDR	k	UWt	No-blocks	Error-check

Table 4: Transmitter Control Packet Format (type 1)

```
struct tstate {
        int     k;
        int     UWt;
        int     No-blocks;
        int     Error-check;
    };
```

Figure 15 : C Definition of Transmitter Control Packet

SNR header is similar to other packets except that the type field contains 1 to indicate that this packet contains the transmitter's state. The next two fields contain the values of the variables k and UWt of the transmitter just prior to the transmission of the packet. Similar to the receiver control packets, k is the interval between two control packet transmissions of the transmitter in units of Tin, updated by machine T3, UWt is the maximum sequence number of the block below which every block has been transmitted, but not necessary acknowledged, No-blocks field contains the number of blocks in the OUTBUF buffer waiting to be sent and this is given by the formula: *(MAXBUF + (TAIL - RETRANS)) mod MAXBUF*

Data packet format: in addition to the SNR_HDR where type field contains 2 to indicate that this packet contains the data, we have the data field whose length is constant throughout the connection, and determined during the connection establishment phase. If a message does not fit into an integral number of packets, the space in the last packet will be padded with null characters. This is intended to simplify the packet processing in the receiver, the block-no extends across the lifetime of the connection, and the packet-no gives the sequence number of the packet within the block. The format of the data packet and its C definition are given in Table 5 and Figure 16 respectively.

SNR_HDR	data	Error-check

Table 5: Data Packet Format (type 2)

```
struct packet {
        u_char          data[DEF_DATALEN];
        int             Error-check;
        };
```

Figure 16: C Definition of the Data Packet

Connection request packet format: contains the negotiated parameters that both the transmitter and receiver must agree on before sending any data. The format and C definition of the connection request packet are shown in Table 6 and figure 17 respectively.

SNR_HDR	mode-communi-cation	peak-bandwidth	packet-size	block-size	receiver-buffer	RTD

Table 6: Connection Request Packet Format (type 3)

```
struct connection {
        int     mode_communication;
        int     peak_bandwidth;
        int     packet_size;
        int     block_size;
        int     receiver_buffer;
        int     RTD;
        };
```

Figure 17: C Definition of Connection Request Packet

In the SNR header, block-no is not used and packet-no field contains the sequence number of the packet, type field is equal to 3, mode-communication field contains 0, 1 or 2, one of the three defined mode of operation of the SNR protocol, peak_bandwidth contains the maximum bandwidth during the connection, the next two fields contain the packet size and block size which remain constant during the connection, receiver_buffer field contains the buffer required at the receiver in units of blocks and since this value is set by the receiver, this field is used only for the conn_ack packet (type 4). The RTD field also is not used in this type of packet. It contains the estimated round trip delay in microseconds,

which is computed by the transmitter (T3) and available only for the conn_conf packet (type 5). The same format of packet is used for type 3, 4, 5 and 6 to make the processing of packets easier.

4. Data Structure

The four transmitter machines, T1 through T4, communicate through global variables in shared memory as explained earlier in this chapter. The structure of this shared memory, called GLOB and pointed to by "gptr" is shown in Figure 18.

```
typedef struct {
        struct sockaddr_in    destination;
        Mesg                  DATA;
        FLAG                  F;
        struct connection     con;
        lup                   LUP[max_lup];
        u_char                mode;
        int                   RTD;
        int                   UWt;
        int                   sock;
        int                   Tin;
        int                   L;
        int                   lci;
        int                   T1_pid;
        int                   T2_pid;
        int                   T3_pid;
        int                   T4_pid;
        } GLOB;
```

Figure 18: Data Structure of the Global Variables in Shared Memory

Mode, of type unsigned character, is initialized by machine T4 and defines the mode of communication of the connection, RTD the round trip delay, estimated by T2, UWt, updated by T1, is the sequence number of the block, below which all blocks are transmitted, sock is the raw socket identifier obtained by T4 during the initialization phase, T_{in} is the period of the timing mechanism of the protocol, L is the maximum window size, lci is the logical connection identifier, T1_pid through T4_pid are the processes identifiers of the four transmitter machines, destination is a system structure that holds the address of the receiving host, passed to T4 when it is first invoked. DATA is a structure that defines the OUTBUF ring buffer and is shown in Figure 19.

```
typedef struct {
        int             RETRANS;
        int             TRANS;
        int             TAIL;
        struct packet   OUTBUF[MAXBUF];
        } Mesg;
```

Figure 19: OUTBUF Ring Buffer Structure

F is a structure of type *FLAG*, shown in Figure 20 and defines the global variables that cause different machines to make their transition and communicate with each other.

Con is a structure of type *connection*, shown in Figure 17 and contains the negotiated parameters, and finally *LUP* is the lup_table with size *max_lup* and contains *lup* structure shown in Figure 21.

T2 and T2_slave communicate the same way using the shared memory and the structure used, called GLOB1 and pointed to by "gptr1," is shown in Figure 22.

```
typedef struct {
        u_char    Transmit;
        u_char    Accept;
        u_char    Fail;
        u_char    T_active;
        u_char    Sent;
        u_char    Disconnect;
        u_char    Scount;
        int       NOU;
        int       Buffer;
        } FLAG;
```

Figure 20: FLAG Structure

```
typedef struct {
        int     seq;
        int     count;
        int     Ack;
        } lup;
```

Figure 21: LUP Structure

```
typedef struct {
        int             s;
        u_char          flag;
        R_CHANNEL       RC;
        } GLOB1;
```

Figure 22: Structure of Shared Memory between T2 and T2_slave

```
typedef struct {
        int         TRANS;
        int         TAIL;
        struct RCH  R_CHAN;
        } R_CHANNEL;
```

Figure 23: R_CHANNEL Structure

```
struct RCH {
        u_char      data[MAXPACKET];
        };
```

Figure 24: RCH Structure

s is the socket identifier needed by T2_slave to receive from the socket, *flag* is a boolean variable when reset to FALSE by T2 causes T2_slave to terminate, R_CHANNEL is a structure that defines the receiver channel (ring buffer). Figures 23 and 24 show this structure.

5. Functions used by Transmitter Machines

In the protocol specification [Ref. 3], subroutines used by both transmitter and receiver were described in the form of algorithms using ADA's syntax. In this section the C source code of these functions, along with other functions needed for the implementation, are given. Access to global variables is made through *gptr*, a global pointer that points to the shared memory.

void Enqueue(P): Inserts the data packet P at the end of the circular buffer OUTBUF.

struct packet Dequeue(): Returns the data packet in the buffer OUTBUF following the TRANS pointer and advances the TRANS pointer to the next location.

int Empty(): Returns true if TRANS = TAIL, indicating that there are no data packets in the transmitter buffer.

int Full(): Returns true if there are no empty buffer locations in OUTBUF to write data packets.In mode 0 true is returned if TAIL +1 = TRANS and in mode 1 and 2, true is returned if TAIL +1 = RETRANS, all operations are modulo MAXBUF.

void Update_OUTBUF(LWr): Advances the RETRANS pointer of OUTBUF so that it points to the buffer location just before the first packet of block number LWr, thus leaving the acknowledged packets out of the retransmission buffer area.

int Expired_LUP(): Returns the sequence number of the first expired block, or if none of the blocks has expired, it returns zero.

void Insert(UWt): After the transmission of a whole block of packets has been completed (block number UWt), this function makes an entry into the LUP table for the block and initializes the retransmission counter. The initial value of the retransmission counter is calculated by the formula: *RTD/Tin + cons.*

void Update_LUP(LOB,LWr,k): Every time a control packet is received from the receiver, the transmitter updates the LUP table using this function. To update the table, the ACK bits of the acknowledged blocks are set to 1, thereby allowing new entries to be made into the table, and the retransmission counters of the acknowledged blocks are decremented by k, which is read from the receiver control packet.

void Balance(LOB,HOLD,LWr,LWt): In mode 1 or mode 2, every time a control packet is received, this function, called by machine T2, decrements the global variable NOU, accessed through the pointer gptr in shared memory, by the number of newly acknowledged blocks. To accomplish this, the bit-map of the previous control packet, which is stored in the variable HOLD, is compared with the LOB field of the currently received control packet.

int Acceptable(con_req): Evaluates the connection parameters in the con_req packet received from the receiver. Returns true if all the parameters are acceptable.

void Enqueue_RCH(P): This function is used by T2_slave to insert the received packet P into the R_CHANNEL buffer.

int Empty_RCH(): Returns true if R_CHANNEL is empty.

struct RCH Dequeue_RCH(): Returns the packet in the R_CHANNEL buffer, following the TRANS pointer and advances the TRANS pointer to the next location.

int Full_RCH(): Returns true if there are no empty buffer locations in R_CHANNEL.

void start_timer(t): Using this function, the internal clock system will send a SIGALRM signal every t microseconds, and the process calling this function must catch this signal.

void stop_timer(): This function causes the timer to stop.

int receiveit(): Returns true if R_CHANNEL buffer is not empty and the packet in the front is of type connection acknowledgment (CON_ACK) or connection request (CON_REQ). It also copies this packet into the con_req global packet of the calling process. Returns zero otherwise. This function is used by T2 during the connection phase.

int receive_state(rs,seq,size): Returns true if the R_CHANNEL buffer is not empty and the packet in the front, which is copied into rs, is of type R_state.

void tvsub(new_time,old_time): Given two "timeval" structure new_time and old_time, this function returns the difference new_time - old_time, used when computing the RTD.

int Checksum(ptr,nbytes): returns checksum of the nbytes bytes beginning at address ptr. The checksum is returned in the low-order 16 bits (the integer is assumed 32 bits).

The C source code of these functions is given below:

```
void Enqueue(P)
struct packet P;
{
 gptr->DATA.OUTBUF[(gptr->DATA.TAIL +1)% MAXBUF] = P;
```

```
 gptr->DATA.TAIL = ((gptr->DATA.TAIL +1)%MAXBUF);
}

 struct packet Dequeue()
{
 gptr->DATA.TRANS = (( gptr->DATA.TRANS +1)% MAXBUF);
 return(gptr->DATA.OUTBUF[gptr->DATA.TRANS]);
}

int Empty()
{
 return(gptr->DATA.TRANS == gptr->DATA.TAIL);
}

int Full()
{
 if (gptr->mode)     /* mode is 1 or 2 */
   return(((gptr->DATA.TAIL +1)%MAXBUF)==gptr->DATA.RETRANS);
 else        /* mode 0 */
   return(((gptr->DATA.TAIL +1)%MAXBUF)==gptr->DATA.TRANS);
}

void Update_OUTBUF(LWr)
int  LWr;
{
  gptr->DATA.RETRANS = (((LWr-1)*gptr->con.block_size)%MAXBUF);
}

int Expired_LUP()
{
 int  i;
 for (i=0 ; i < max_lup; i++ )
   if ((gptr->LUP[i].Ack ==0) &&(gptr->LUP[i].count ==0))   /* found it */
     return(gptr->LUP[i].seq);
```

```
 return(0);   /* in case of the whole table was looked and conditions not satisfied */
}
void Insert(UWt)
u_char  UWt;
{
 gptr->LUP[((UWt -2)%max_lup)].seq = UWt;
 gptr->LUP[((UWt -2)%max_lup)].count = (gptr->RTD/gptr->Tin) + cons;
 gptr->LUP[((UWt -2)%max_lup)].Ack = 0;
}

void Update_LUP(LOB_ptr,LWr,k)
int  LWr , k ;
u_char  *LOB_ptr;
{
 int i;
 /* set the Ack of acknowledged blocks to 1 */
 for (i=0 ; i < max_lup ; i++ )
   if (gptr->LUP[i].seq < LWr )
     gptr->LUP[i].Ack =1;
 for (i=0 ; i < size_LOB ; i++ )
   if ((*(LOB_ptr +i)) == 1 )
     gptr->LUP[((LWr +i - 2)%max_lup) +1].Ack =1;
 /*  decremente counters of unack blocks */
 for (i=0 ; i< max_lup ; i++ )
   if (gptr->LUP[i].Ack == 0 )
     if (gptr->LUP[i].count <= k)
       gptr->LUP[i].count = 0;
     else
       gptr->LUP[i].count = gptr->LUP[i].count - k;
}
void Balance(LOB_ptr,HOLD_ptr,LWr,LWt)
u_char     *LOB_ptr, *HOLD_ptr;
int        LWr,LWt;
{
```

```
 int  i;
 for (i = LWt ; i < LWr ; i++ )
   if ( !(*(HOLD_ptr + i +1 - LWt)) )
     gptr->F.NOU--;
 for (i = LWr ; i < (LWt + gptr->L) ; i++ )
   if ( (*(LOB_ptr + i  - LWr +1 )) && !(*(HOLD_ptr + i - LWt + 1)) )
     gptr->F.NOU--;
 for (i = (LWt + 1 - LWr + gptr->L) ; i <= gptr->L ; i++ )
   if ( (*(LOB_ptr + i ) )
     gptr->F.NOU--;
}

int Acceptable(con_req)
struct connection con_req;
{
 if ((con_req.mode_communication  != gptr->con.mode_communication)  ||
     con_req.peak_bandwidth       != gptr->con.peak_bandwidth)       ||
     con_req.packet_size          != gptr->con.packet_size)          ||
     con_req.block_size           != gptr->con.block_size)           ||
     con_req.receive_buffer       != gptr->con.receive_buffer)       ||
     con_req.RTD                  != gptr->con.RTD)                  || )
         return(0);
 return(1);  /* the else part  */
}

void Enqueue_RCH(P)
struct RCH  P;
{
 gptr1->R_CHAN[(gptr1->TAIL +1)%MAX_CHAN] = P;
 gptr1->TAIL = ((gptr1->TAIL +1)%MAX_CHAN);
}

int Empty_RCH()
{
```

```
 return(gptr1->TRANS == gptr1->TAIL);
}

struct RCH  Dequeue_RCH()
{
 gptr1->TRANS = (( gptr1->TRANS +1)% MAX_CHAN);
 return(gptr1->R_CHAN[gptr1->TRANS]);
}

int Full_RCH()
{
  return(((gptr1->TAIL +1)%MAX_CHAN)==gptr1->TRANS);
}

void  start_timer(t)
int  t;
{
 if (t < 1000000 )
    {
     itv.it_interval.tv_sec = 0;
     itv.it_interval.tv_usec = t;
    }
 else
    {
     itv.it_interval.tv_sec = t/1000000;
     itv.it_interval.tv_usec = 0;
    }
 itv.it_value = itv.it_interval;
 setitimer(ITIMER_REAL,&itv,(struct itimerval *)0);
}

void stop_timer()
{
 itv.it_interval.tv_sec = 0;     /* itv is a itimerval structure variable in "snr_tr.h" file */
```

```
 itv.it_interval.tv_usec = 0;
 itv.it_value = itv.it_interval;
 setitimer(ITIMER_REAL,&itv,(struct itimerval *)0);
}

int receiveit()
{
 struct RCH   rpack;
 register struct SNR_HDR  *snr;  /* pointer to SNR header  */
 register struct connection  *uptr;  /* start of the ACK packet */
 if ( Empty_RCH() )
   return(0);
 rpack = Dequeue_RCH();    /* the else part */
 snr = (struct SNR_HDR *) rpack.data;
 uptr = (struct connection *) (snr + SIZE_SNR_HDR);
 if((snr->lci ==gptr->lci)&&((snr->type ==CON_ACK)||(snr->type==CON_REQ)))
   {
   con_req.mode_communication = uptr->mode_communication;
   con_req.peak_bandwidth = uptr->peak_bandwidth;
   con_req.packet_size = uptr->packet_size;
   con_req.block_size = uptr->block_size;
   con_req.receive_buffer= uptr->receive_buffer;
   con_req.RTD = uptr->RTD;
   return(1);
   }
 return(0);
}

int receive_state(rs,seq,size)
struct rstate  *rs;
int  *seq;
int  size;
{
 struct RCH   rpack;
```

```
register struct SNR_HDR   *snr;
register struct rstate   *uptr    /* start of the receiver state info */
if ( Empty_RCH() )
  return(0);
rpack = Dequeue_RCH();
snr = (struct SNR_HDR *) rpack.data;
uptr = (struct rstate *)(snr + SIZE_SNR_HDR);
if ((snr->lci ==gptr->lci)&&(snr->type ==R_state))
  {
   rs->k = uptr->k;
   rs->LWr = uptr->LWr;
   rs->buffer_available = uptr->buffer_available;
   bcopy(uptr->LOB,rs->LOB, size);
   rs->Error_check = uptr->Error_check;
   (*seq) = snr->seq.packet_no;
   return(1);
  }
 return(0);
}

void tvsub(out,in)
register struct timeval  *out;
register struct timeval  *in;
{
 if (( out->tv_usec -= in->tv_usec) < 0)   /* subtract microsecondes */
   {
    out->tv_sec --;
    out->tv_usec += 1000000;
   }
 out->tv_sec -= in->tv_sec;    /*  subtract seconds  */
}

int Checksum(ptr,nbytes)
register  u_short  *ptr;     // pointer to the first 2 bytes we need to compute the checksum
```

```
register   int         nbytes;  // total number of bytes we need to compute the checksum
{
 register long  sum = 0;  //  holds the accumulated sum
 u_short        oddbyte;   // used when nbytes is odd
 register  u_short  answer;  // holds the returned value( u_short assumed 16 bits)
 while(nbytes > 1 )
 {
  sum  +=  *ptr++;
  nbytes  -= 2 ;    // since ptr pointes to 2 bytes at a time
 }
 if(nbytes == 1)  // takes care of case where nbytes is odd
 {
  oddbyte =0;
  *((u_char *) &oddbyte) = *( u_char *) ptr;
   sum  += oddbyte;
 }
// add back carry outs from top 16 bits to low 16 bits
sum  = ( sum >> 16) + ( sum & 0xffff);  // add high-16 to low-16
sum  += (sum >> 16);  //  add carry
answer  = ~sum;  // ones-complement, then truncate to 16 bits
return( answer );
}
```

C. MACHINE IMPLEMENTATIONS

In this section a C style algorithms for different transmitter processes are given. Algorithms follow closely the state diagrams and the Predicate Action Tables introduced in the specification of the protocol. Also, the fifth process, T2_slave, added to the protocol, is presented in more details.

1. T2_slave Process

As mentioned earlier, this process is added to the protocol for implementation reason. Invoked by T2, it is responsible for only receiving packets from the raw socket,

and enqueuing them into the R_CHANNEL. It executes this function as long as the variable "flag" remains true (it is reset to FALSE by T2).

The T2_slave algorithm is:

```
local variable declaration;
attach shared memory GLOB1 to space address;
set signal SIGUSR1 to be correctly handled;  // this signal is needed because
    // the receivefrom is a blocking system call, that can be interrupted, and
    // when it is time to quit, T2 deblocks it by sending this signal.
STATE = 0;
while(STATE != 2)   // when it is time to quit, STATE is set to 2
 switch(STATE)
 {
  case 0 :
        if(flag ==T)
          STATE =1;
        break;
  case 1 :
        if((flag==T)&&(there is packet in raw socket))
        {
         strip off the IP header from this packet;
         enqueue the resulting SNR packet into R_CHANNEL;
        }
        if(flag==F)
          STATE =2;
        break;
  }     // end case
// STATE is 2  so quit
detach shared memory and exit;
```

2. Process T1

At the end of the connection phase made by T2, and if this phase is successful, process T1 is forked by process T4. Initializes its local variables, attaches the shared memory GLOB to its address space and begins executing following the structure of the FSM diagram and the PAT specification described in Chapter II.

The T1 algorithm is given below:

```
local variables declaration and initialization;
UWt =1; retrans_count =1; trans_count =1;
attach the shared memory GLOB to address space (accessed through gptr);
open Sentsem, Nousem and Lupsem  semaphores;(they are created by T4)
STATE =0;
while(STATE != 4)  // when it is time to quit,STATE is set to 4
 switch(STATE)
 {
 case 0 :
        if(T_active==T)
          STATE =1;
        break;
 case 1 :
        if((T_active==T)&&(mode==2)&&(Expired_LUP())) //retrans.
        {
         index = (((Expired_LUP()-1)*b_size)+retrans_count);
         block_no = Expired_LUP();
         packet_no = retrans_count;
         packet = OUTBUF[index%MAXBUF];
         add SNR_HDR to the packet;
         compute checksum;
```

```
 send it  to raw socket using sendto system call;
 acquire Sentsem semaphore;
 Sent = T;
release Sentsem semaphore;
 retrans_count++;
 if(retrans_count > b_size)
 {
   retrans_count =1;
   LUP[(Expired_LUP()-1)%L +1].count = Initial_value;
  }
 }
if(T_active   &&
  ! Empty()  &&
  (trans_count <= b_size)  &&
  (  (mode ==0)  || ((NOU<L)&&((Buffer-NOU)>0)&&
       ((mode==1)|| !Expired_LUP())   )))
 {
 retrans_count =1;
 block_no = UWt;
 packet_no = trans_count;
 packet = Dequeue();
 add SNR_HDR to this packet;
 compute checksum;
 send it to raw socket using sendto system call;
 acquire Sentsem semaphore;
 Sent = T;
 release Sentsem semaphore;
 trans_count++;
```

```
        }
        if(T_active ==F)  // quit
            STATE = 4;
        if(trans_count > b_size)  // block compleated
        {
         trans_count =1;
         UWt++;
         copy UWt to global UWt;  //used by T3 in T_state packet
         STATE =2;
        }
       break;
case 2 :
        if(mode==0)  // no flow control
          STATE =1;
        else   // mode is 1 or 2, so flow control
        {
          acquire Nousem semaphore;
          NOU++;
          release Nousem semaphore;
          STATE =3;
         }
        break;
case 3 :
        if(mode ==2)   // error check
            acquire Lupsem semaphore;
            Insert(UWt);
            release Lupsem semaphore;
        STATE =1;
```

```
        breack;
    }   // end case
    // STATE is 4, so quit
    detach the shared memory,close semaphores  and exit;
```

3. Process T2

T2 is the first process "forked" by T4. T2 initializes its local variables, attaches the shared memory to its space address,"forkes" T2_slave to begin receiving from the raw socket, and executes its function in accordance with the specification of chapter II.

T2 process algorithm is given below:

```
    local variables declaration;
    reset =10; max_attempts =3;  // used for the connection phase
    sequence =1;  // receiver control state_packet number
    attach GLOB to space address;
    get and attach shared memory GLOB1 ( used with T2_slave )
    flag =1;  // 1 means TRUE, this for T2_slave to begin( flag is in GLOB1)
    fork T2_slave;
    open Scountsem,Nousem  and Lupsem  semaphores;
    sighold(SIGALRM)  // ignore this signal for now
    set the SIGALRM signal to be handled //for the clock_tick implementation
    start_timer(DEFAULT_Tin);
    STATE =0;
    while(STATE != 8 )  // 8 means quit
     switch(STATE)
     {
     case 0 :
          if(Transmit==T && Accept==T && Fail==F )
```

```
        {
         timer_flag =0 // it will be put to 1 when SIGALRM arrives
         sigrelse(SIGALRM) //now the timer will pop every Tin usec
         gettimeofday(old_t)  // used to compute RTD
         send the con_req type packet to raw socket;// using sendto call
         STATE =1;
        }
        break;
case 1 :
        if(receiveit())  //  the R_CHANNEL(front) = conn_req
        {
         stop_timer();
         if( Acceptable(con_req)) // receiveit() dequeue into con_req
         {
          gettimeofday(new_t);  // new_t - old_t is RTD
          tvsub(new_t,old_t);  // the difference is in new_t structure
          RTD = new_t.tv_sec*1000000 + new_t.tv_usec;
          Tin = RTD/kou ;
          T_active = T;
          send a CON_CONF packet to the receiver;
          STATE =2;
         }
         else   // unacceptable
         {
         Accept = F;
         STATE = 8;
         }   // end else unacceptable
         send SIGUSR2 to T4   // T4 waits to be awaken by this signal
```

```
        }   // end if receiveit()
        else   // we don't receive a packet
          if( timer_flag ==1 )  // clock  case(SIGALRM was arrived)
           {
            timer_flag = 0   // reset it so next time we can detect SIGALRM
           delay++;
           STATE = 6;
           }
        break;
case 6 :
        if (delay < reset)   // OK  case
           STATE = 1;
        else      // timeout  case
         {
           attempts++;
           delay =0;
           STATE =7;
         }
        break;
case 7 :
        if (attempts < max_attempts)  // retry  case
        {
          gettimeofday(old_t) ;  // computing RTD begins at this time
          send another CON_REQ packet;
         STATE =1;
         }
        else   // quit
        {
```

```
        stop_timer();
        FAIL = T;
        STATE = 8;
        send SIGUSR2 to T4;
        }
      break;
case 2 :
      if((receive_state(rs,sequence,sizeof(LOB)))&&(Disconnect==F))
        STATE = 3;
      if(Disconnect = T)   // abort  case
      {
        T_active = F;
        Transmit = F;
        STATE = 8;
      }
      if((Transmit==F) &&
         Empty()  &&
         (Disconnect==F) &&
         (((mode==2) && Empty_LUP()) || (mode != 2)))   // finish
       {
         T_active =F;
         send a DISC_TYPE packet;
         STATE =8;
         }
      break;
case 3 :
      if ( sequence > high )  // update  case
      {
```

```
        acquire Scountsem semaphore;
        Scount =0;
        release Scountsem  semaphore;
        high = sequence;
        STATE =4;
       }
      else  // seq >= high  so discart it
        STATE =2;
      break;
case 4 :
      if( mode==0)  // no flow control
        STATE =2;
      else   // mode is 1 or 2, flow control
       {
        acquire Nousem semaphore;
        Balance(rs.LOB,HOLD,rs.LWt,LWt);
        release Nousem  semaphore;
        save rs.LOB into HOLD;
        LWt = rs.LWr;
        Buffer = rs.buffer_available;
        Update_OUTBUF(LWt);
        STATE = 5;
       }
      break;
case 5 :
      if(mode==1)  // no error check
        STATE =2;
      if(mode==2)  // errorcheck
```

```
        {
         acquire Lupsem semaphore;
         Update_LUP(HOLD,LWt,rs.k);
         release Lupsem  semaphore;
         STATE = 2;
        }
 }  // end switch
// case 8,quit
flag =0;  // used by T2_slave
send SIGUSR1 signal to T2_slave;
detach shared memory, close semaphores  and exit
```

4. Process T3

Upon successful connection phase, T2 sends a signal (SIGUSR2) to T4 process, which looks at global variables set by T2 in shared memory to see the result of the connection phase. At that time,T3 process will be *forked* by T4 if the connection is established. The following is a C style algorithm of T3:

```
declare local variables;
STATE =0;      // defines the FSM of T3
count =0;     // used to ...
UWt =1;        // used to update the global variable UWt
attach shared memory;
open Sentsem, Scountsem  semaphores;
initialize SNR_HDR;
set signal SIGALRM to be handled properly when it arrives;
hold this signal for now;
```

```
while( STATE != 6)       // STATE will be set to 6 to quit
 switch(STATE)
 {
  case 0 :
          if(T_active is TRUE)     // start
          {
           start_timer(Tin);  // now the timer will pop each Tin usec
           STATE =1;        // move to the next state
           }
           break;
  case 1 :
          if(T_active is FALSE)    // quit
          {
           stop_timer();
           STATE = 6;
           }
          else
           {
            sigpause(SIGALRM);  // wait for clock-tick to arrive
            acquire Scountsem semaphore;
            increment Scount;
            release Scount  semaphore;
            STATE =2;  // move to state 2
           }
          break;
  case 2 :
          if(Sent is FALSE)    // no data has been transmitted
          {
```

```
        increment count;
        STATE = 3;    // move to state 3
       }
      else    // T1 has sent data,so send control packet
      {
        snr_hdr->seq.packet_no++;  // increment sequence no
          //   prepare fields of control packet to send
        control_packet.k = k;
        control_packet.UWt = global_variable UWt;
        control_packet.No_blocks=(MAXBUF+(TAIL-
                                    RETRANS))%MAXBUF
        send control packet using sendto system call;
        k =1;
        STATE =4;
       }
      break;
case 3 :
      if((count < k)&&(Scount < Lim))  // delay
        STATE =1;
      else   // timeout
       {        // send control packet with same parameter as case 2
        increment sequence no of control packet;
        update control packet(k,UWt,No_blocks);
        send it to the raw socket using sendto system call;
        k = MAX( 2*k , k*Lim));
        STATE = 4;
       }
      break;
```

```
case 4 :
        if(Scount < Lim)     // no disconnection
        {
          acquire Sentsem semaphore;
          Sent = FALSE;
          release Sentsem  semaphore;
          count =0;
          STATE =1;
         }
        else    // disconnect
         {
           Disconnect = TRUE;
           STATE =5;
          }
        break;
case 5 :    // to quit we wait until T_active, set by T2, becomes FALSE
       if( T_active is FALSE )    // quit
          STATE =6;
       break;
}   // end case

detach shared memory and exit;
```

5. Process T4

T4 is the first process invoked in the protocol. In the current implementation we assume the following:

When a user wants an SNR service, he invokes T4 by issuing the command "T4 destination", where destination is the name of the host which we want to connect to, data to be transferred are assumed in a file named "my_file" and each line of this file holds a complete data packet.

The C style algorithm of T4 is:

```
local variable declaration;
create shared memory GLOB using shmget system call;
attach GLOB to address space using shmat system call;
create Sentsem, Scountsem, Nousem and Lupsem semaphores;
transform destination name to internet address;
store this address in the global variable destination;
get the raw socket identifier using socket system call;
initialize global variables, including negotiated parameters;
set signal SIGUSR2 to be correctly handled;
open the file "my_data", data packet to be sent are in this file;
fork process T2;
wait T2 to establish connection, SIGUSR2 will arrive;
// at this point the signal has arrived
STATE =1;
while(STATE != 0)    // 0 when we finish
 switch(STATE)
  case 1 :
        if(Fail is TRUE)     // quit
        {
         Transmit = FALSE;
         STATE = 0;
        }
```

```
        if(Accept is FALSE)     // quit
          STATE = 0;
        if(T_active is TRUE)    // start
        {
         fork T1  and T3;
         STATE = 2;
        }
        break;
case 2 :
        if(T_active is FALSE)   // disconnect
          STATE =0;
        else
            if( no data to transmit)   // finish
            {
             Transmit = FALSE;
             STATE =3;
            }
            else                // write
                if(  ! Full())
                 {
                  read one packet;
                  Enqueue(this packet);
                  }
        break;
case 3 :
        if(T_active is FALSE)
          STATE =0;
}  // end case
```

detach shared memory, close semaphores and exit;

IV. CONCLUSIONS AND RECOMMENDATIONS

This thesis describes the implementation of the transmitter part of the SNR protocol (Transport protocol for high speed networks), made up with four different machines working in parallel, using C programming language. The implementation of the receiver part is done in parallel in a separate thesis. The implementation was done on top of the Internet Protocol (IP) using three workstations running two different versions of Unix Operating System (Solaris and Irix) and connected with FDDI network, allowing up to 100 Mbps throughput.

The structure of the C source code of each machine reflects the Finite State Machine and the associated Predicate Action Table as presented in the specification of the protocol. The conversion of the logic of the machines to C language environment is made straight forward by a very clear specification, whose correctness was proved in previous thesis work.

Dealing with the operating system functions efficiently in order to improve the overall speed of the execution of the protocol was our main goal in our implementation.The four transmitter machines were implemented as four Unix processes and therefore the Inter-Process Communication (IPC) was an important issue in this implementation. The four different processes use shared memory to communicate with each other, which provides the best way in Unix facilities. An SNR packet is transmitted and received using the raw socket, a form of IPC provided by Unix Operating System.

A fifth process is added to the transmitter protocol to interface with the IP layer by receiving packets from the raw socket and enqueuing them into R_CHANNEL.

Some corrections and implementation details are added to the protocol. In Figure 6 and Figure 10, either one of the two conditions *count* = *k* or *scount* = *Lim* is sufficient for the transition *timeout* to occur, in contrary to the specification, where the *and* is wrongly used instead of *or*.

In order to send all types of data (including binary files), another field, data_length (16 bits) is added to the SNR packet format, allowing the detection of the end of the file. Also to guarantee the termination of the protocol without loosing the last packets, a problem that is overlooked in the specification, the block-number and packet-number of the last packet are sent with the disconnect packet.

Initially in the specification, only three different types of packet are defined, R_state, T_state and data packets. Four other types are added to the protocol in this implementation, necessary for the connection establishment and termination, Con_req, Con_acq, Con_conf and Disc_type packets.

Some of the problems we had in implementing the SNR protocol are:

- The signal facilities provided by Unix were unreliable in earlier versions, meaning that race conditions existed and signals could get lost [Ref. 16]. Enhancements are made in both versions under which we implement the SNR, to provide reliable signals. The problem is these enhancements are not compatible and consequently, it was more difficult to ensure the portability of the code.

- Since the number of processors is limited, the operating system decides which one of the five processes to run first, using a scheduling algorithm. This random behavior makes difficult the choice of the correct parameters of the protocol. These parameters must be set so that even if we do not receive packets from the receiver for a long period, it does not mean necessarily that the receiver is dead and it can be caused by a scheduling problem.

- The T_{in} interval of the protocol depends on the Round Trip Delay (RTD) which is computed based on the system clock. The precision of this clock is limited and for high speed networks, RTD can not be correctly computed.

Some tests were conducted on the protocol (transmitter and receiver) in all three modes, using both versions of the Unix operating system (Solaris and Irix). These tests were necessary to choose the correct parameters. The transmitter sends text files that can be either found in a special directory or entered by the user. We used three types of text

files, small (less than 100 bytes), medium (between 100 bytes and 10 kbytes) and large files (more than 10 kbytes). Packet sizes were 20, 28, 40 and 72 bytes, and 8 packets per block. Errors are made by not sending or unacknowledged some packets, the lost packets were eventually retransmitted by the transmitter. In each state of the protocol, statements are printed on the screen showing the progress of the protocol. More details on the implementation tests can be found in [Ref. 23].

In this thesis we showed that the SNR protocol can easily be implemented in software using the current workstations.

Further improvement of this implementation could concentrate on:

- Implementing the SNR on top of the data link layer directly rather than using the IP layer, which we expect results in a less overhead.

- Implementing the SNR in a multiprocessor environment (five processors for the transmitter and five for the receiver), where each machine is dedicated to a different processor.

- Make the existing applications (FTP, Email...) run with the SNR rather than TCP.

- Using multicasting, so the same transmitter can send data to more than one receiver at a time.

- And finally testing and comparing the implementation with other transport protocols such as TCP/IP.

LIST OF REFERENCES

1. Gerard J.Holzman,Design and validation of Computer Protocols,Prentice-Hall,Inc, 1991.

2 Lundy,G.M.,Miller,R.E.,Specification and Analysis of a Data Transfert Protocol Using Systems of Communicating Machines,Distributed Computing 1991.

3 Gilbert M. Lundy, Tipici,H.,Specification and Analysis of the SNR High Speed Transport Protocol, IEEE/ACM Transactions on Networking, Vol. 2, No. 5, October, 1994.

4 High,W.,a Formal Protocol Test Procedure for the Survivable Adaptable Fiber Optic Embedded Network(SAFENET),Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

5 Leffler,S.,Mckusick,M.,Quarterman,J.,the Design and Implementation of the 4.3 BSD Unix Operating System,Addision-Wesley Publishing Company, 1989.

6 Kelley,A.,and Ira,P., A Book on C,second edition,the Benjamin/Cummings Publishing Company,Inc, 1990.

7 Marc,J.R.,Advanced Unix Programming,Prentice-Hall,Inc, 1985.

8 Matthew,J.,R.,a Tool for Automated Validation of Network Protocols,Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1992.

9 Maurice,J.,B.,the Design of the Unix Operating System,Prentice-Hall,Inc,1986.

10 Misha,S.,Thomas,F. La Porta,Performance Analysis of MSP:Feature-Rich High-Speed Transport Protocol,IEEE,Transactions on Networking vol.1,#6,December, 1993.

11 Morten,S.,Implementation of Physical and Media Access Protocols for High-Speed Networks,IEEE,Communication Magazine,June, 1989.

12 Netravali,A.,Roome,W.,and Sabnani,K.,Design and Implementation of a High Speed Transport Protocol,IEEE,Transactions in Communications,vol.38,#11,November, 1990.

13 Olav,K.,Applications of High-Speed Networks,Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1991.

14 Olah,A.,L.,and S.M.Heemstra de Groot,Design and Implementation Issues of End-to-End Protocols in High-Speed Networks,Tele-Informatics and Open Systems(TIOS) 1993.

15 Partridge,C., Gigabit Networking,1994.

16 Stevens,W.,R.,Unix Network Programming,Prentice-Hall,Inc,1990.

17 Sharon,H.,and Dan,S.,Analysis of Transport Measurements Over a Local Area Network,IEEE,Communications Magazine,June, 1989.

18 Stallings,W.,Data and Computer Communications,Fourth edition,Macmillan Publishing Co.,1994.

19 Thomas,F.,La Porta,Architecture,Features and Implementation of High-Speed Transport Protocols,GLOBECOM'91,IEEE 1991.

20 Thomas,J.,D.,Design and Implementation of a Fiber Optic Link for a Token Ring Local Area Network,Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1992.

21 Clark,D.,Jacobson,V.,Romkey,J. and Salwen,H.,An Analysis of TCP Processing Overhead,IEEE,Communications Magazine,June, 1989.

22 W. J. Wan, Implementation of the SNR High-Speed Transport Protocol (the Receiver Part),Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1995.

23 R. Grier,Test of the SNR Implementation,Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr G. M. Lundy, Code CS/Ln 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Colonel-Major Ezzeddine Dkhil 1
   Direction de l'Informatique et de Statistique
   M.D.N, 1030 Bab Mnara, Tunis/ TUNISIA

6. Lt-Colonel Med Bechir Echockl 1
   Direction de l'Informatique et de Statistique
   M.D.N, 1030 Bab Mnara, Tunis/ TUNISIA

7. Commandant Mezhoud Farah 1
   Direction de l'Informatique et de Statistique
   M.D.N, 1030 Bab Mnara, Tunis/ TUNISIA

8. Mr le Directeur 1
   Direction du Personnel et de Formation
   M.D.N, 1030 Bab Mnara, Tunis/ TUNISIA

9. Colonel Sellami 1
   Direction des Transmissions
   Quartier Bab-Saadoun, Tunis/ TUNISIA